Fault Injection Experiment Results in Space borne Parallel Application Programs[1]

Raphael R. Some Raphael.R.Some@jpl.nasa.gov, Won S. Kim Won.S.Kim@jpl.nasa.gov, Garen Khanoyan Garen.Khanoyan@jpl.nasa.gov, Leslie Callum Leslie.N.Callum@jpl.nasa.gov, Anil Agrawal Anil.K.Agrawal@jpl.nasa.gov, John J. Beahan John.J.Beahan@jpl.nasa.gov , Arshaluys Shamilian Arshaluys.Shamilian@jpl.nasa.gov , Allen Nikora Allen.Nikora@jpl.nasa.gov
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
(818) 354-1902

Abstract—Development of the REE Commercial Off The Shelf (COTS) based space-borne supercomputer requires a detailed knowledge of system behavior in the presence of Single Event Upset (SEU) induced faults. When combined with a hardware radiation fault model and mission environment data in a medium grained system model, experimentally obtained fault behavior data can be used to: predict system reliability, availability and performance; determine optimal fault detection methods and boundaries; and define high ROI fault tolerance strategies. The REE project has developed a fault injection suite of tools and a methodology for experimentally determining system behavior statistics in the presence of application level SEU induced transient faults. Initial characterization of science data application code for an autonomous Mars Rover geology application indicates that this code is relatively insensitive to SEUs and thus can be made highly immune to application level faults with relatively low overhead strategies.

TABLE OF CONTENTS

1. INTRODUCTION

The objective of the Remote Exploration and Experimentation (REE) Project is to bring supercomputing technology into space. It has twin goals of:
1. demonstrating a process for rapidly transferring commercial Commercial-Off-The-Shelf (COTS) high-performance computing technology into ultra-low power, fault tolerant architectures for space and
2. demonstrating that high-performance onboard processing enables a new class of science investigation and highly autonomous remote operation.

The REE project is employing mainly (COTS) hardware and software components, and relying on Software-Implemented Fault Tolerance (SIFT) to mitigate the effects of radiation-induced errors. Natural space radiation can cause soft errors known as Single Event Upsets (SEU's) in non-radiation-hardened electronics. Thus, REE's primary reliability concern is the detection and mitigation of SEU's.

To provide ease of mission insertion, flexibility in configuration, straightforward upgrade as the state of the art progresses, and for ease of fault tolerance insertion, REE's architecture of choice is a cluster computer. The intent is to leverage the considerable technology investments made in commercial cluster computers and to augment or modify the standard commecial cluster architecture as necessary to provide enhanced reliability for the embedded spaceborne environment. A cluster computer consists of interconnected stand-alone computers working together as a single integrated computing resource. Some of the salient characteristics of cluster computers are: multiple high performance processors with local memory, fast network interconnects, high-bandwidth/low-latency communication protocols, a standard Operating System (OS) on each processor, access to shared mass storage and a convenient parallel programming environment.

Figure 1 shows the baseline REE architecture comprising a set of processing and mass memory nodes which are multiply interconnected via a high speed switched network fabric. Each processing node consists of two state of the art processors, eg. PPC750's or G4's, a large local memory and a network interface. In addition to network low level interconnect functions, the network interfaces also provide high level I/O handling protocols such as Message passing Interface (MPI) so that the node processors may be off-loaded from relatively mundane I/O tasks. The mass memory nodes provide a large (several GigaBytes) non

[1] U.S. Government work not protected by U.S. Copyright.

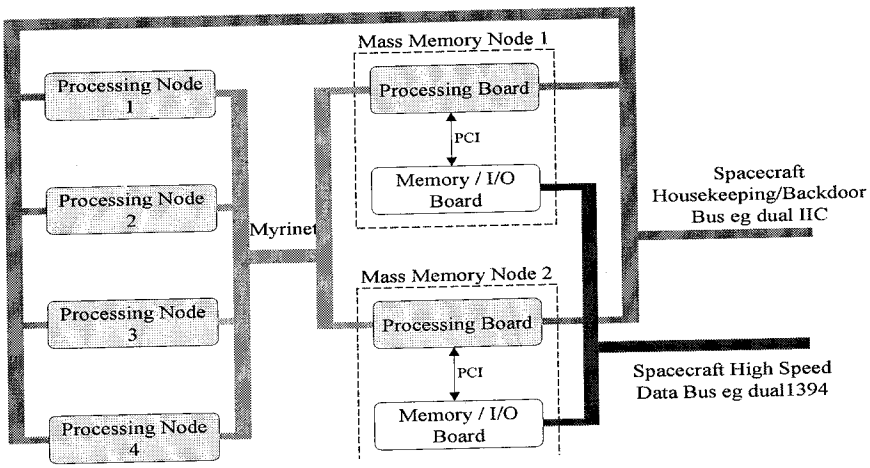

Figure 1: REE Cluster Computer Baseline Architecture

volatile memory for the system. Multiple such "disk emulators" are provided for parallel high speed access by the processing nodes as well as for fault tolerance support. The spacecraft interfaces to the REE computer via these mass memory nodes through the inclusion of a custom I/O controller in each mass memory node, thus providing mulitiply redundant interconnections to the spacecraft data and housekeeping busses. The spacecraft control computer and instrument controllers view the REE cluster as a mass memory device. Instrument data and processing commands are written, as files, to the mass memory while processed data and REE status are accessed as file read opeations. The spacecraft housekeeping bus is extended to the individual nodes of the REE computer to facilitate externally commanded diagnostic proceedures by the Spacecraft Control Computer (SCC). It is both noteworthy and serendipitous that advanced processor architectures are increasingly implementing low level fault detection/protection mechanisms and some of these mechanisms are baslined into the REE architecture. These include Single Error Correct Double Error Detect (SECDED) Error Detection and Correction (EDAC) on local memories, parity protection on external caches, exception detection in Arithmetic and Logic Units (ALUs) and Memory Management Units (MMUs), and watchdog-configurable timers.

The development of an effective and efficient fault detection and mitigation strategy for REE requires a detailed knowledge of fault: types and rates, propagation paths, behaviors and probabilities. These characteristics are dependent on the hardware used to implement the architecture, the radiation environment, the system and application software, and their interactions. In order to investigate these issues, a method utilizing experimentation to obtain fundamental effects, and modeling to predict system level behavior, performance, reliability and availability has been developed. The following section explains the overall REE experimentation and modeling methodology and tool set.

2. Methodology Overview

The REE project requires a means for trading off performance and power utilization versus reliability and availability. The method must be generally applicable to alternative architectures and applications and, once developed, relatively straightforward to implement. Unlike traditional fault tolerant systems, a degree of unreliability or unavailability, i.e., .95 or .99 rather than .99999 is often an acceptable reliability figure for REE applications. On the other hand, it is imperative that the system fault behavior and reliability be accurately predictable. The mission system engineer must be able to 'dial in' a desired level of reliability and fault behavior based on mission phase and criticality. Thus, a methodology is required which will allow characterization and modeling of probabilistic system behavior, reliability and availability under varying applications, environments, loads, and operational scenarios.

Figure 2 shows the methodology and tool set developed for the REE project and is explained below:

Radiation effects experiments are performed on the hardware components to determine subsystem level radiation sensitivities. Results for a processor, for example, include fault rates for the L1 data cache, the L1 instruction cache, the General Purpose Registers (GPRs), the Floating Point Registers (FPRs), the (MMU), etc.

The results of the radiation experiments are used to develop a radiation fault model. This model is used to predict the fault rates that will occur in a given radiation environment (e.g., Low Earth Orbit, Geosynchronous Orbit, Deep space, Solar Flare, etc.). The model provides the number of faults per unit time per subsystem.

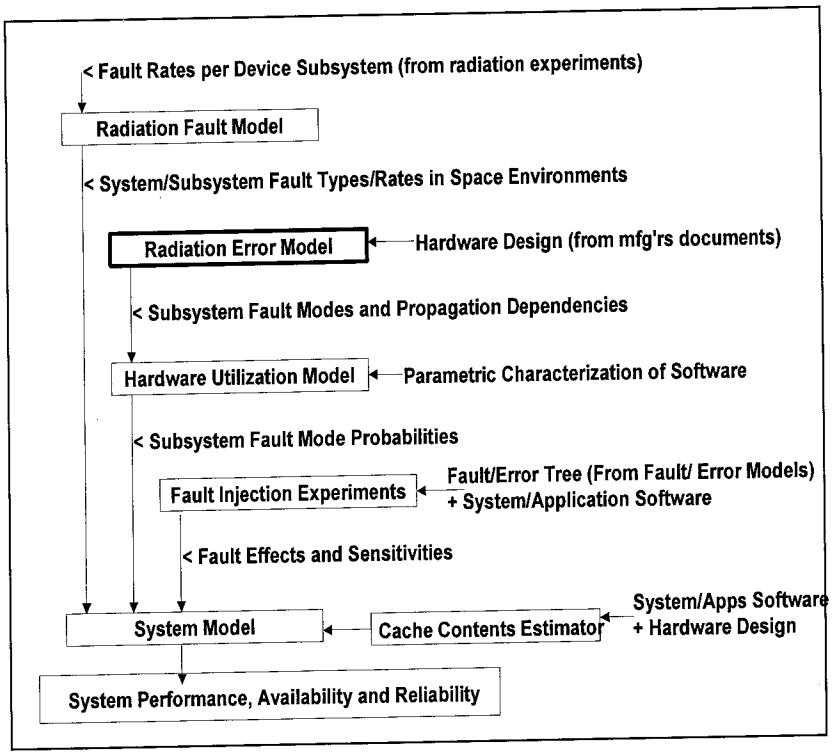

Figure 2: Overall REE Methodology Block Diagram

of an SEU occurring in a given subsystem. Essentially, the process of generating the error model is one of listing all possible faults and then, by analysis, propagating each fault through the hardware to the first point at which it impacts software or system operation. The emphasis of this effort is on subsystems into which faults cannot be directly injected with Software Implemented Fault Injection (SWIFI). Thus, it is not necessary to trace every possible error resulting from a general -purpose register bit flip. It is however, necessary to list all the possible outcomes of SEUs in MMU and cache address translation registers, cache tag rams, etc.

The Hardware Utilization Model, provides a means for determining the software (hardware utilization) dependent probabilistic fault propagation statistics and the method by which SWIFI fault injection techniques can be used to emulate the effects of the underlying faults enumerated in the Error Model.

The central component of this methodology is the construction and execution of fault injection campaigns. Fault injection campaigns are designed to provide fault/error sensitivities of the system components. The campaigns are conducted on the operational system, after which the results are analyzed to determine the effects of the faults (e.g., system crash/hang, incorrect result, no apparent effect) and their associated probabilities.

The Cache Contents Estimator (CCE) is used to deal with the inability of SWIFI techniques to inject bit flip faults into the processor's cache memories. It is, in effect, as special case of the Error Model/Hardware Utilization Model explained above. Faults are injected into an application's instruction, data, heap, and stack segments in main memory to determine the fault behavior statistics of each type of error. The CCE predicts how much (and which portions) of each of these segments will be in the cache at any given time. The final error rate for each of these segments in cache is proportional to its size in unprotected L1 Cache.

Finally, the system reliability and performance model is constructed using knowledge of the system architecture, predictions from the fault model, the results of the fault injection experiments and the CCE results. The model predicts the system's reliability and performance in a given radiation environment. It can be used during system development to identify appropriate system architectures and fault tolerance strategies. During fielded operation, the model can be used to predict the system's behavior in changing circumstances and modify it as appropriate (e.g., increase checkpointing frequency, uplink fault-tolerant linear algebra libraries). Once the basic system model has been created and validated, it is relatively straightforward to input application software specific fault behavior statistics, input the mission environmental parameters, and predict system fault behavior and reliability for a range of fault tolerance techniques, thus it also provides an early quick look resource for mission development.

3. JIFI FAULT INJECTION TOOL SET

The JPL Implementation of a Fault Injector (JIFI) tool set allows fault behavior characterization of an application. It allows automated campaigns of random uniformly distributed bit-flip-faults into application registers and/or memory space. Injection can be targeted to an application, a set of related applications or to specific routines within an application. Code, stack, heap or data may be targeted for memory injection and specific registers, register sets, or all registers may be targeted for register fault injections.

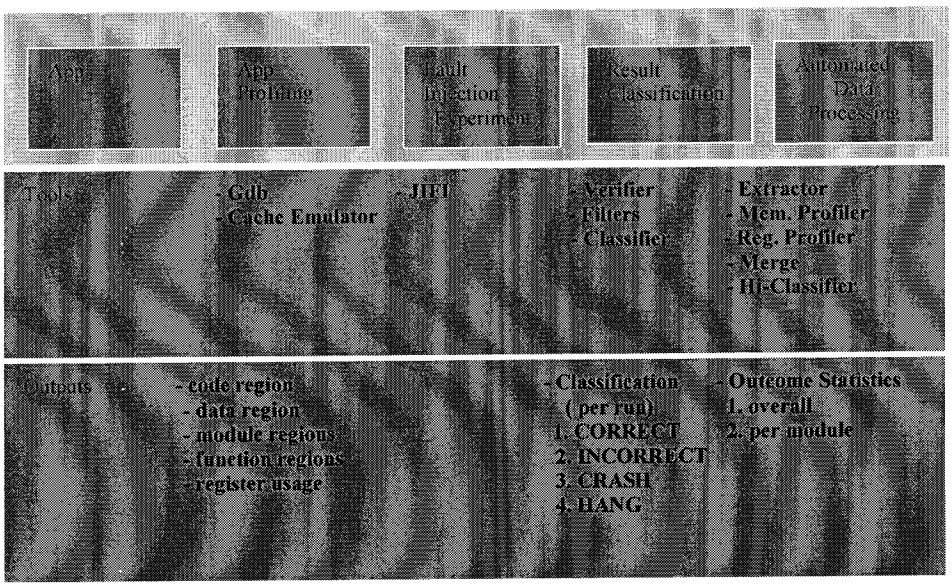

Figure 3: Fault Injection Tool Set

In the case of targeted fault injection, it is necessary to know the range of memory addresses and registers one wishes to inject. Profiling the application provides this information. In the case of random program-wide injection, it is also useful to be able to map the faults back to where/when they were injected, i.e.,
What was the program executing when the fault was injected?
What was at the memory/register location the fault was injected into?
Application profiling provides the information necessary for post-injection campaign reconstruction as well.

To characterize the fault injection results, it is necessary to verify the final result and classify its effect on the system, i.e., correct output, incorrect output, process crash/hang.

The overall output of this tool set and methodology is to provide overall SEU fault sensitivity of the application or portions thereof:
Is the application more likely to crash or produce incorrect results?
Which subroutines of the program are most vulnerable?

The JIFI fault injection tool set provides the application profiling, fault injection and results verification and classification as defined above as well as statistical processing of campaign and multi-campaign data. These tools are written in either C or Perl scripting language. Figure 3 shows the tools and their usage.

The GNU debugger (GDB) is used to get a dump of the program's subroutines and global data addresses. This operating system available tool enables the user to relate fault and program counter location to actual subroutines of the application. Another application profiling tool is the cache emulator, this tool provides the ability to monitor how the L1 cache behaves as a stream of instruction and processor main memory references are executed. This tool basically simulates cache activity and provides information on how much of the program is in L 1 cache at any point in its execution or on average.

The JIFI tool provides the fault injection experiment capability. JIFI is an application-level fault insertion tool that is used with a user defined configuration file and one or more execution scripts in order to produce batches of runs (campaigns). Following a single injection run, it produces an output file containing: program counter value at injection time, address injected into, time of injection, and the application state upon end of the run.

Several tools are used for result classification. A verifier of some sort must be available to compare the output file produced by an fault injected run to that produced by a run with no fault injection (gold-run output file). A simplistic, first order verifier does a binary compare of the two files with no tolerance for insignificant errors. For many applications however, it is more useful to have a verifier that takes into account the system noise characteristics and

required or useable accuracy. This type of "smart verifier" is usually provide by the application programmer. The JIFI output file will have this information appended to it. Based on the correctness thresholds within the verifier the application output file is labeled as correct, incorrect, or incomplete.

The filter tool is applied to the JIFI output files to remove any runs considered bad from the batch of good runs to analyze. A run may be considered bad if a fault was not injected (this occurs rarely, but is possible due to the random injection method and poor timing resolution of some systems) or if the output file was corrupted in some way by the fault injected (this can occur if the injection boundaries allow injection into areas of application space used by the JIFI fault injector or other non-application code).

The classify tool determines the classification of each fault injection run based on input from JIFI and the application verifier. This information is read from the JIFI output files of each node and compared against a rule definition file that defines the classification for all possible combination of outcomes. Based on this, a run of the application will be classified as correct, incorrect, crashed, or hung. This result is appended to the JIFI output file.

Finally the automated data processing is handled by a small set of tools. The extract tool gathers multiple run data into a single data file. The resulting file can then be easily imported into such applications as MS Excel and MS Access for further analysis. The Virtual Memory Profiler tool labels the memory and Program Counter (PC) location of each fault injected with the software module component and subroutine. The memory profiler requires JIFI output file and application subroutine and global data information from the application profiling to produce its results. There is also a register profiler tool, which gives statistics on the register usage for a portion of code. This tool gives a count of how many times each register has been seen in GDB a disassembly file. This tool provides knowledge about the register usage of an application and therefore gives further insight on the classifications statistics seen in register injection. The merge tool combines the outputs of the Extract tool with the output of the Virtual Memory Profiler

into a single file and sorts the runs according their classification. It also displays
summary information that can be captured to a file. The High Level Classifier allows grouping multiple experiments. It consists of two functional components, hl_class_mem and HL_Class_Reg. hl_class_mem sorts the combined data in the order of software module components and then sorts them in the order of classification for any type of memory injection. It also gives statistics on each software module component according to classification. HL_Class_Reg is used to sort the combined data in the order of register numbers and then sort them in the order of classification. It also gives statistics on each register according to classification.

Figure 4 showes that the tools are used in sequence to come up with a characterization of the application. This process can be captured in a high level processing script that can automate processing batches of data. Following each run, the application output file is verified against its gold file. The verification result is appended to the output file. After a batch of runs are completed there will be x number of output files (1output file for each run times the number of nodes the application is running on). These JIFI output files are run through the filter to weed out the bad injection runs and put them in a separate directory. Once filtering is complete each run is classified and the final classification information is appended to the JIFI output file. Extract will pull out the user specified JIFI output fields of each run into a file. This

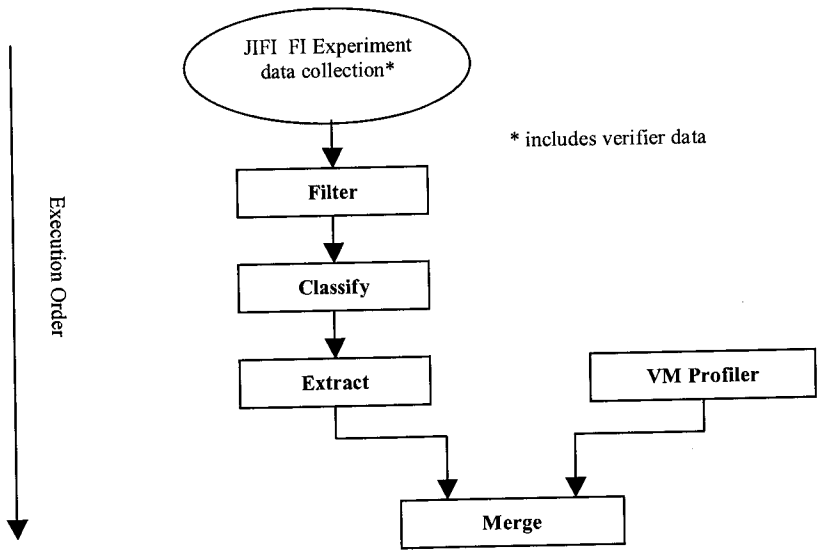

Figure 4: Data Processing Procedure

file will be brought together with the application virtual memory profile by the merge tool. At the next level the results of different experiment batches can be put together using High Level Class to get combined characterization data.

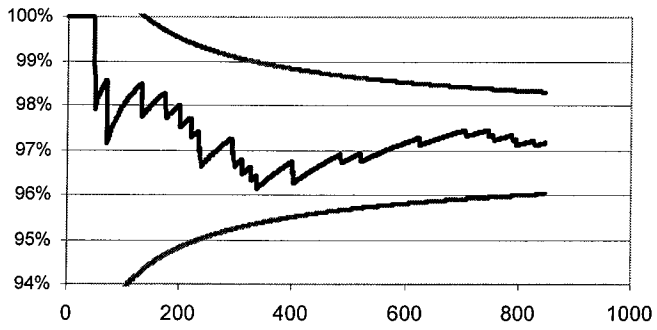

Figure 5: Statistical error tests for Correct Runs

There is one final item in the JIFI tool kit. The Statistical Significance Estimator is used to estimate the confidence level provided by the campaign. It generates the +/- 2 Sigma curves against a plot of the running average of result behavior (incorrect, crash/hang., etc) vs. the number of injections. If the running average is well behaved and confined within the 2σ curves, we can estimate the confidence level of the experiment as a function of the number of injection experiments. If the plot is not well behaved, it is an indication that there are some variables (e.g., different input images) that cause the fault injection outcome to be statistically non-stationary. For most of the "controlled" experimental campaigns we have run to date, with the same data input condition (e.g., the same input image) for each run, approximately 1000 injections are required to reach a reasonable (95%) confidence level. Figure 5 shows an example of a statistical error test using the texture segmentation program. It shows the "moving" average vs. the number of correct runs, obtained from 1000 fault injection runs. For each run, a fault was injected into the entire application code region. The surprising result is how well behaved these campaigns have been and how few injection experiments have been required (approx. 1000 to 1500 injections per fault type per fault area) and how closely "global level" results have tracked a weighted average of "modular" injections. The following section defines Global and Module level fault campaigns, their usage and construction as well as providing results from a series of campaigns.

4. Fault Injection Campaign Construction

The goal of the fault injection campaign is to elucidate the behavior of the application program in response to the types of faults it may encounter due to SEU's (i.e., radiation induced single and multi-bit transients or "soft errors") in the operational environment. What is ultimately desired is a statistical distribution of fault behaviors for each type of fault in each region of the application. Fault type, in this context, refers to the number and location of bits flipped, and application region refers to a region of code and its associated data, stack and heap areas. In addition, fault injection campaigns are also used to determine sensitivity of a system or system component to various fault types and rates. Finally, the traditional use of fault injection, i.e., to test a system for fault coverage, can also be effected with a fault injection campaign.

For our purposes, we have defined two types of fault injection campaign: Global and Module.

In Global fault injection, the entire application is treated as a black box. Its internal structure is not profiled and faults are injected randomly throughout its memory and register space. This type of fault injection is the easiest and least expensive to perform and often provides sufficient data for initial reliability estimation. For many mission applications, this is sufficient and no further fault injection experimentation is required. If we are attempting to understand the probabilistic behavior of a system to SEU's, then poisson statistics would dictate that we inject faults throughout the system randomly in space and time. If, on the other hand, we are attempting to determine system fault coverage, then it is preferable to ensure that the fault is present and active at system execution time. In the case of code and data segment errors, for example, it is useful to pre-inject the error and then start the system. For heap or stack faults, however, this strategy will not work as the error maybe overwritten prior to use – in these cases, massive campaigns of random injection may be necessary.

Module fault injection entails more detailed profiling of the application and its component routines followed by targeted fault injection into the various software components. This type of fault injection is useful for a number of activities:

- development of detailed fault sensitivity maps
- determination of optimal fault detection boundaries and mechanisms
- development of fault tolerance programming guidelines
- cataloging of fault effects/sensitivities of reusable software such as libraries
- development and validation of fault tolerant reusable software libraries and usage guidelines

Like global fault injection, local fault injection can be done randomly throughout the execution time or can be more narrowly focused in time to ensure that the fault will be activated depending on the goal of the campaign.

Interesting issues to examine with combined local and global fault injection campaigns include:
Determining if time weighted local random statistic match random global statistics for a given application program

Determining local error behavior vs. global behavior correlation with respect to the severity of the erroneous behavior, e.g., an error in an Fast Fourier Transform (FFT) may lead to an unacceptably incorrect system output or it may have no discernable effect. Developing cost effective analysis and characterization tools methods and tools based on local fault injection of library components, for example, is a long-term area of study, which might provide a significant pay off.
Determination of latent fault, and correlated fault sensitivities and effects.

To date, we have not dealt with issues 1 through 4 above with the exception of a cursory look at weighting local vs. global statistics per 1 above. Further, to date we have mainly performed single fault injections. For purposes of the REE project (i.e., given the mission environments of interest, the expected fault rates and reliability requirements), these experiments were deemed to be sufficient for initial system analysis and modeling. Future work may include additional experimental work to further investigate issues1-4 above.

Mars Texture Application Fault Injection Campaigns

The following provides an example of a series of fault injection campaigns used to investigate the behavior of a proposed scientific data processing package for autonomous geological exploration of Mars. The function of the application is the classification of rock type by textural features.

The fault injection experiments detailed herein were run on the Mars Texture Segmentation program running under the Lynx operating system on a Motorola PPC604 cluster. The texture segmentation program is written using MPI supporting parallel computing on multiple nodes. For these experiments only two nodes were used to run the application. The program reads a rock image. The image is first transformed by performing an FFT. The transformed image is then filtered to generate three intermediate images by iterating a gabor filter and inverse FFT three times with different filtering parameter. A 2-level k-means classifier labels each image pixel either 0 or 1 based on the three filtered intermediate images. The resultant output file is a "segmented image" where segmentation is determined by the texture values assigned to each pixel. The texture indicates whether the pixel is part of the rock or a background region. The segmented or labeled image is saved into a file. The total run time for this program is about 20 seconds. On the Motorola PPC604 cluster using 2 nodes, the FFT routine takes about 0.5 s, and Gabor filtering 0.2 s as shown in. The inverse FFT calls the same FFT function as the forward FFT, and takes the same amount of time. The k-means classifier takes about 4 s, or in some cases, about 8 s if the solution does not converge in the first try. Writing the output label file takes about 10 s. The total run time is approximately 20 s. The output label file is needed during the data analysis phase to classify the outcome of a fault injection. Figure 6 shows the structure and execution timeline of the application.

For the campaigns discussed herein, the output verifier used was a simple binary compare. The classifier provides the following possible classifications:
Correct: Output label file was created and matches with Gold file
Incorrect: Output label was created but incorrect
Crash: Output label file was not created and JIFI output reported application crash
Hang: Application was timed out by JIFI after the maximum time limit

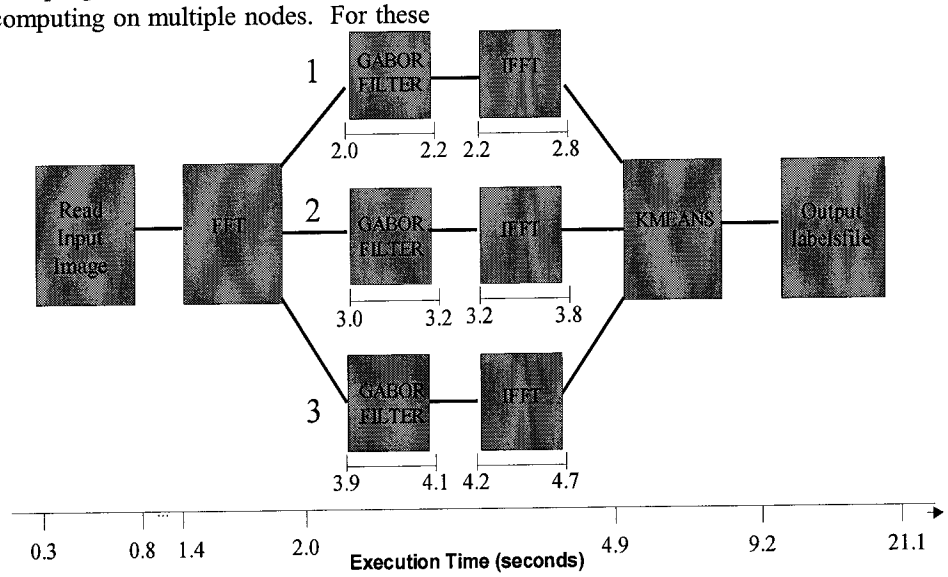

Figure 6: Functional block diagram and timeline of the texture segmentation program

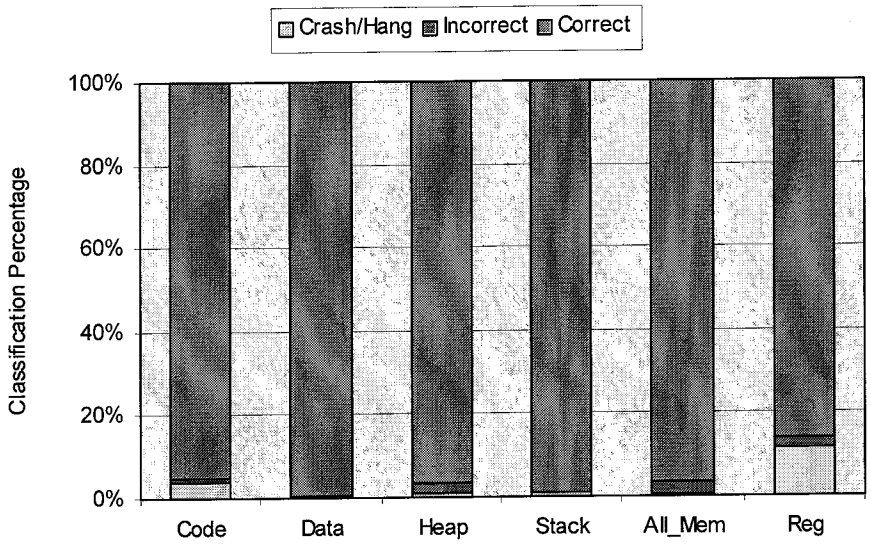

Figure 7: Global Fault Injection into Application Virtual Memory and GPR

Overall Application Fault Behavior Statistics

To determine the application's contribution to the system's overall failure rate, reliability and availability, a global random fault injection campaign was constructed. The results of this campaign can be combined with fault arrival information from the radiation fault model and system performance estimated with a system model in which is embedded the system fault tolerance strategy and behavior. For this campaign 3000 single bit faults were injected into each of the application memory code, stack, data and heap regions and into all general purpose and floating point registers. Faults were injected randomly in time and space. One fault was injected per application execution run. The application was reloaded for each new run to eliminate fault accumulation and latent faults. The results of this campaign are shown in Figure 7.

Figure 7 shows the results (i.e., crash, hang, incorrect output, correct output) of a random fault injection into each memory segment and the cpu register set. The first, obvious conclusion is that some 95% of all faults do not result in erroneous behavior, i.e., they have no effect, thus the program is relatively robust even without any added fault protection code. The percentage of Correct runs for memory regions exceeds the percentage of Correct runs in GPRs. This is clearly due to the probability of injecting a fault into a critical area of memory at a critical time vs the probability of injection into a critical register at a critical time. It is also clear that the heap dominates the experimental results due to the fact that the heap region occupies 94.1% of the all memory regions and that faulty behavior is dominated by erroneous output. Similary, it is clear that erroneous behavior from injections into code result in crashes and hangs but not erroneous output. The results of these experiments also show that the percentage of Correct runs in FPRs injection (not shown in Figure 7) is high. This is because there is a large number of insignificant bits used in these calculations. Post-experiment follow up investigation showd that the number of incorrect runs due to inection in all FPR bits is approximately the same as the number of incorrect runs due to injection into sign and exponential bits. Thus, the Incorrect runs are due to sign and exponential bits faults. It is clear that even this initial experiment can be used later to optimize the system fault protection strategy.

Detailed Fault Sensitivity Studies

The second fault campaign set was constructed to study the sensitivity of each module to faults in its memory and register regions (note that not all registers are used by any given software module). In this set of experiments, a single bit fault was injected per application run and runs were reloaded between executions to flush out latent faults. The

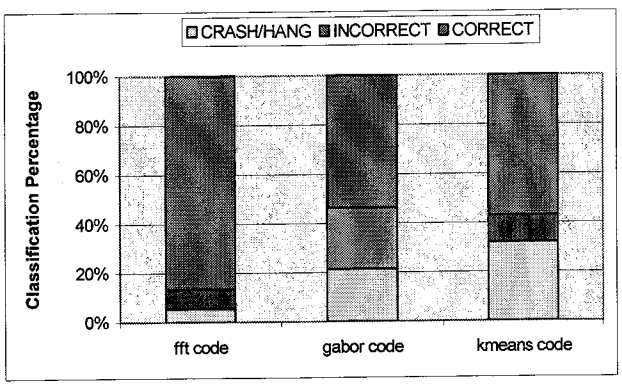

Figure 8: FFT time injecting to FFT code, Gabor time injecting Gabor code, Kmeans time injecting into Kmeans code

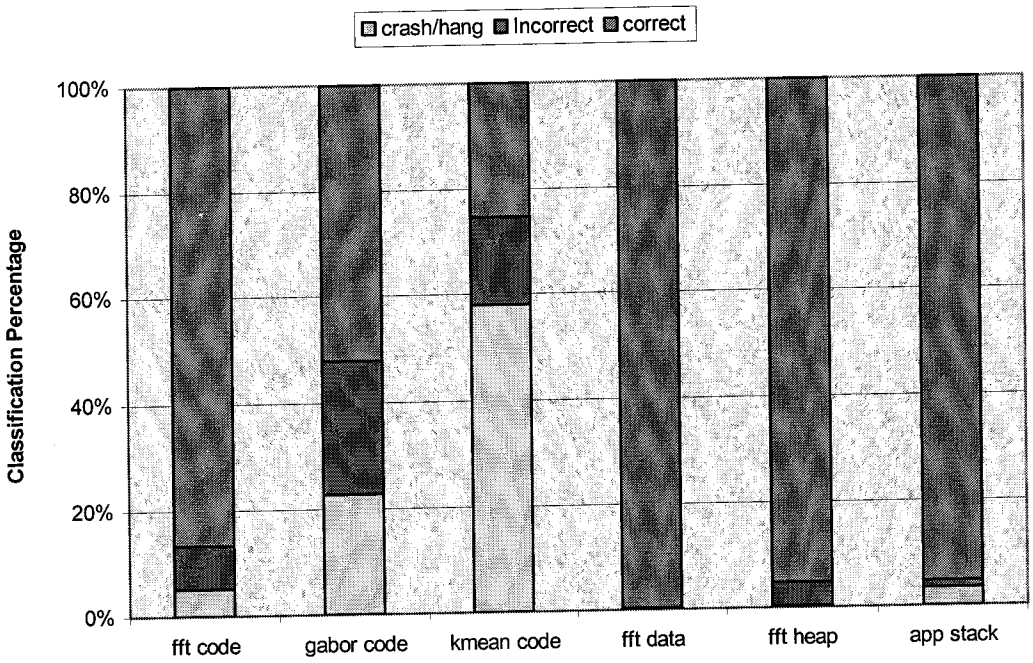

Figure 9: Modular Fault Injection into App. Virtual Memory during FFT Time

faults were injected randomly throughout the memory and register regions used by the module (e.g., FFT, IFFT, k-means, Gabor), during module execution. Figure 8 shows typical the results of this campaign.
Figure 8 shows the sensitivities of different modules. The obvious conclusion of the sensitivities of the modules is that Gabor code is more sensitive than k-means code and FFT code.

A third fault injection campaign was constructed to study the effect of fault latency and to get a more complete understanding of the sensitivity of code and data segments to single bit faults. In this campaign faults were inserted prior to execution of the module. Faults were single bit and were inserted randomly into module data, stack, heap, code and registers used by the module. The Gabor and k-means code regions of Figure 9 show typical results of this campaign.

As can be seen, Gabor and k-means code have greater percentage of crash/hang and incorrect than the FFT code as the faults are injected prior their execution.

A worst case average for module sensitivity, (and a reasonable upper bound for purposes of system analysis) can be estimated by adding the results of latent fault injection for module code and data segments from to the results of random fault injection for module stack and heap. In this case the module is guaranteed to experience all code and data faults and has a reasonable probability of experiencing heap and stack faults.

Figure 9 is an artificial test constructed to illustrate the above. Figure 9 graphs the results of an experiment in which faults were randomly injected throughout the application during FFT execution time. As seen in the figure, Gabor and k-means code injection experiments yield significantly higher failure rates than FFT code injections for reasons explained above. FFT heap segment also resulted in a high failure probability as expected from this injection procedure. Note that FFT data segment injections did not result in a high failure probability due to the fact that this data is static and therefore random injections (like random code injections), minimize the probability of introducing an erroneous datum (instruction) during module execution.

4.4 System Level Fault Tolerance Estimation and Validation

For those applications or mission phases in which guaranteed correct results with bounded real time operation is required, a Triple Modular Redundant (TMR) or, in some cases, Quad Modular Redundant (QMR) system configuration can be use. Assuming a fault rate which is large compared to program execution (or system cycle) time, the probability of such a system experiencing a failure is astronomically high. A system was designed in which a software implemented Byzantine resilient QMR core was used to provide a reliable system management function which, in turn implemented a temporal TMR fault tolerance strategy for critical applications. In the temporal TMR system implemented for this investigation, the application program is run three times in succession on the same nodes. The results of each run are stored as named files in system mass memory. After three executions of the application

Table 1: Example Fault Injection Campaign for Validation

Experiment	Total Number of Runs	Region	Number of Nodes	Number of Faults	Number of Fault Injected Runs per Job
Nominal operation without any fault & no JIFI.	1000	None	1	0	3
Application execution without any fault & JIFI.	1000	None	1	0	3
Application execution with Single Fault injected in registers.	1000	Registers (GPRs)	1	1	3
Application execution with Single Fault injected in application code.	1000	Code	1	1	3
Application execution with Single Fault Injected in application heap.	1000	Heap	1	1	3
Application execution with Single Fault injected in application stack.	1000	Stack	1	1	3
Application execution with Single Fault injected in application data.	1000	Data	1	1	3
Application execution with Multiple Faults injected in application code.	1000	Code	1	2	3
Application execution with Multiple Faults injected in All Memory.	1000	Memory	1	2	3
Application execution with Multiple Faults injected in registers.	1000	Registers (GPRs)	1	2	2
Application execution with Multiple Faults injected in memory and registers.	1000	Memory and Registers	1	2	2
Application execution with Multiple Faults injected in memory and registers.	1000	Memory and Registers	2	2	3

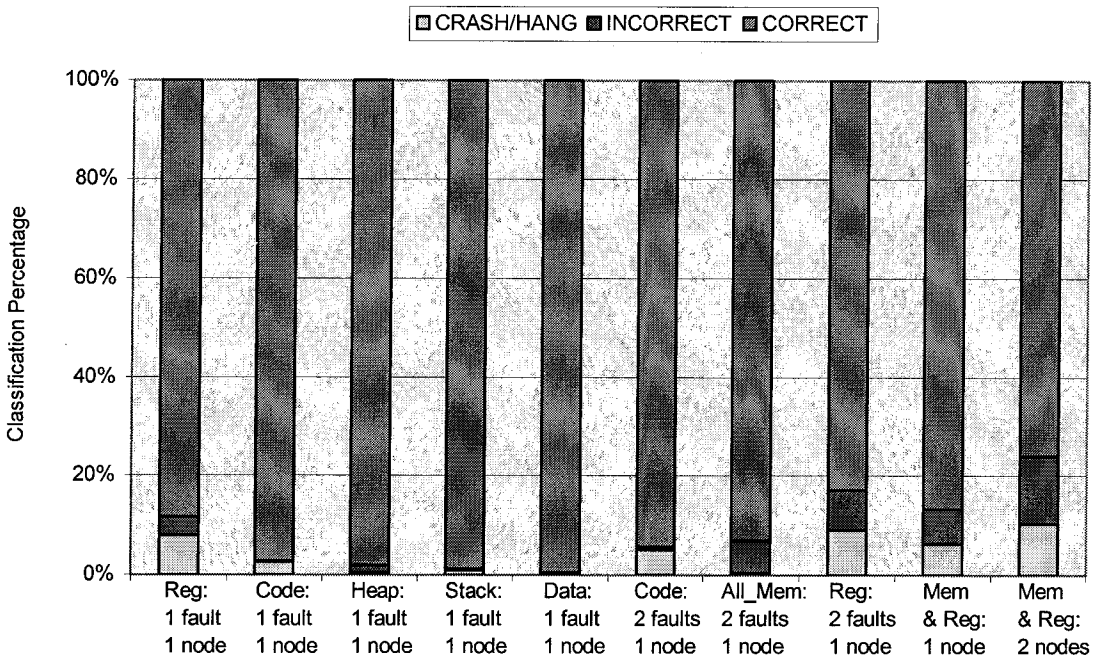

Figure 10: Experiment Results

program, the results are voted by the reliable QMR core. The results are classified as successful or failed. The application execution is declared to be unsuccessful (failed) if:

- No labeled output file is created due to hang or crash for all the three executions
- All three labeled files are different
- Only one label file is generated
- Two different label files are generated

Table 2: Job Level Results of Fault Injection Campaign from Table 1

Type of Experiment (See Table 1)	Number of Correct Label Files out of 999 Runs	Number of Successful Jobs	Number of Failed Jobs
No JIFI	999	333	0
JIFI, no faults	999	333	0
Reg: 1 fault, 1 node	921	322	11
Code: 1 fault, 1 node	985	333	0
Heap: 1 fault, 1 node	982	333	0
Stack: 1 fault, 1 node	990	333	0
Data: 1 fault, 1 node	998	333	0
Code: 2 faults, 1 node	950	328	5
All_Mem: 1 node	924	328	5
Reg: 2 faults, 1 node	835	311	22
Mem & Reg: 1 node	872	322	11
Mem & Reg: 2 nodes	641	237	96

To validate the above system, a campaign was designed to inject both single and multiple faults into the application processors, to independently assess the correctness of the output against a "gold run" with the standard binary verifier, and to then determine the correctness of the QMR core voter output. In the process, the global fault injections were repeated and their results compared to the initial global fault injection campaign results. In addition, it was desired to collect statistical data which could be used in a system model to determine, for the mission environment, the overall system reliability where reliable operation is defined as providing a correct answer within the maximum allocated time, i.e., the time required for the system to execute 3 runs, vote and relay the result.

The campaign consisted of the experiments listed in Table 1:

Figure 10 shows the run level results. From Figure 10, results of this campaign, as expected, are similar to those of the original global fault injection campaign described in 4.2.

Table 2 shows the job level results. In all cases, the voter results were correct. The data from Table 1 was then used to provide state transition probabilities to a system level Markovian model which was in turn driven by the environmental fault rates predicted for the mission environment. Section 5 explains the system model.

5. System Reliability Model

A discrete-state, discrete-time Markov model was developed to predict the behavior of the system in the presence of SEUs in specific radiation environments. The construction of the model, model results, and assumptions made in constructing the model are described in the following paragraphs.

Model Description

The basic building block of the reliability model is shown below in Figure 11. The system starts by setting up

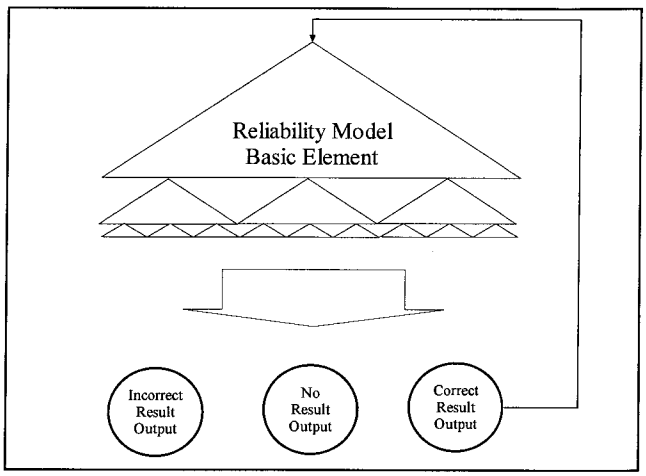

Figure 11: Reliability Model Basic Element

the application for its next run. During that run, a fault may or may not be injected into the application. If a fault is injected into the application, the application may either crash or hang, produce an incorrect result, or exhibit no observable effects of the fault. If no fault is injected, the application is assumed to operate correctly (see "Assumptions" below in this Section) and produce a correct result. The probabilities of encountering a fault during a single execution of an application in a given radiation environment are computed from the fault rates for that environment, assuming that the arrival of faults is described as a homogeneous Poisson process (see "Assumptions" below). The probabilities that an injected fault will result in

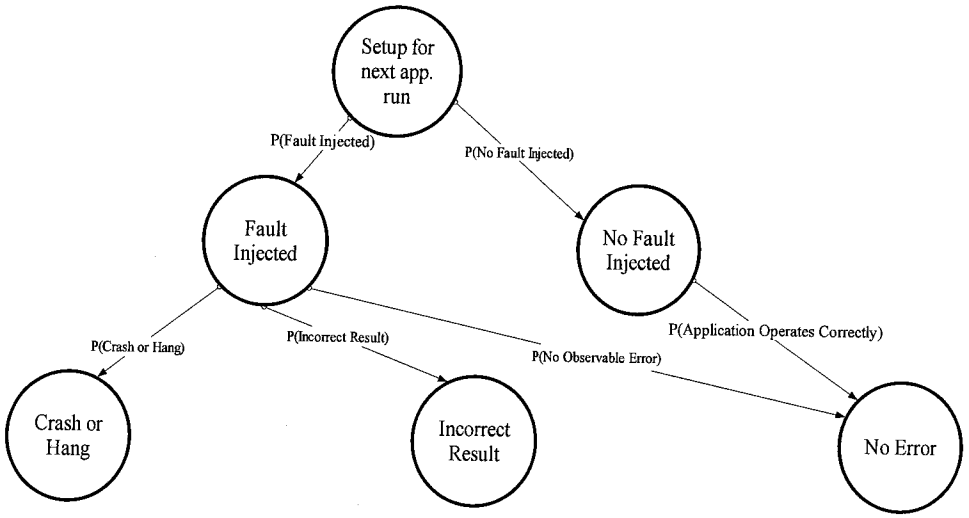

Figure 12: System-Level Reliability Model

the application crashing or hanging, producing an incorrect result, or experiencing no apparent effect, are determined from the results of the fault injection experiments.

For the fault injection campaigns conducted for REE, the application was run under a simple fault tolerant executive. The executive ran the application three times, after which the results were voted. If two of the three runs of the application produced identical results, that result was accepted as correct. Given the radiation environments for representative space missions, this would almost always allow only correct results to be output. Incorrect final results would only be produced if two of the three runs encountered faults and suffered identical effects leading to incorrect output. If, on the other hand, two matching results could not be found, no final result would be produced. This would be the situation if two out of three runs encountered a fault and crashed or hung, if one run crashed or hung and another produced an incorrect result, or two runs encountered faults and produced different sets of incorrect results.

The system-level model is constructed from the basic element as shown in Figure 11. The triangle in Figure 12 represents the basic element shown in Figure 11. The application runs once, after which it can crash or hang, produce incorrect output, or experience no fault effect. Regardless of the ending state of the first run, the application is reloaded and run again (represented by the second row of triangles in Figure 12) after which it is reloaded and run for the third and final time (third row of triangles in Figure 12). After the application has run three times, the results of the three runs are voted. The voting algorithm will either produce a correct final result, an incorrect final result, or no final result. For the purposes of the reliability, the latter two states are considered to be failure states.

The final model was fairly simple, having only 81 states. However, it was useful in showing at a high level how the system would behave in different radiation environments.

Model Results

The system reliability model was used to estimate the system's behavior in a variety of radiation environments. We observed a wide range of behavior – in some environments, the estimated probability of the system's surviving for a 5-year period was over 95%. This is shown in Figure 13, representing the way in which the system's reliability changes over a 300 weeks period in a deep space environment at Solar minimum with no solar flares. On the other hand, the estimated probability of the system surviving for one day without experiencing a radiation-induced error in the case of a solar flare was substantially less than 0.1. The way in which the system reliability changes over a 20 hours period in this environment is shown below in Figure 14.

Assumptions and Model Limitations

The reliability model assumes the following system characteristics and environmental conditions:

1. The occurrence of faults follows a Poisson distribution. The mean of the distribution is obtained from the radiation testing results.
2. In computing the probability of encountering a fault during a run, the probability of encountering one or more faults during that run is computed. For low fault rates, this is very close to the probability of encountering exactly one fault during the run. For higher fault rates, this provides a more accurate characterization of the environment in which the system will be operating.
3. The effects of multiple faults in one run are assumed to be independent of one another.

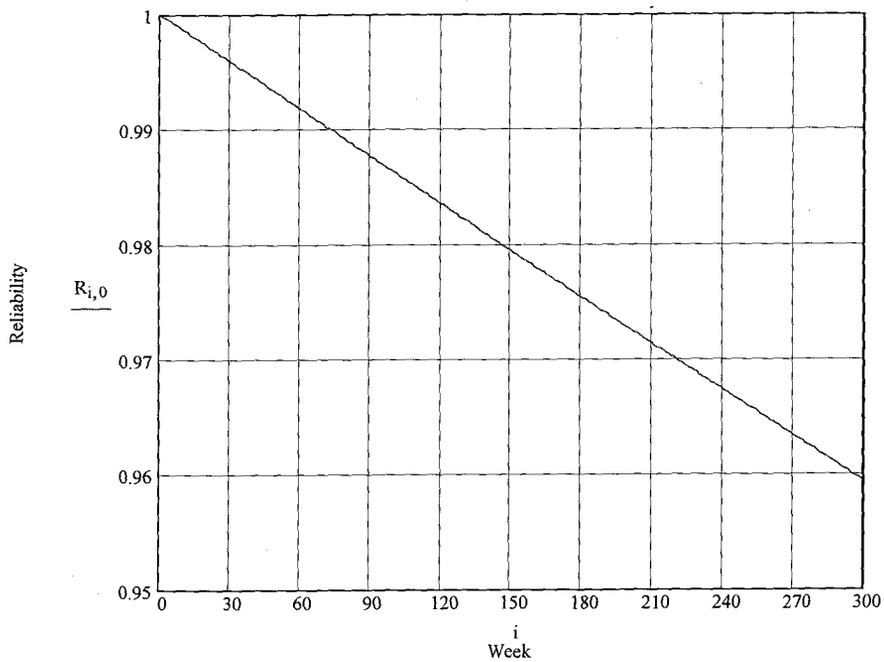

Figure 13: Estimated Reliability Over 300 Weeks for Deep Space Environment, No Solar Flare

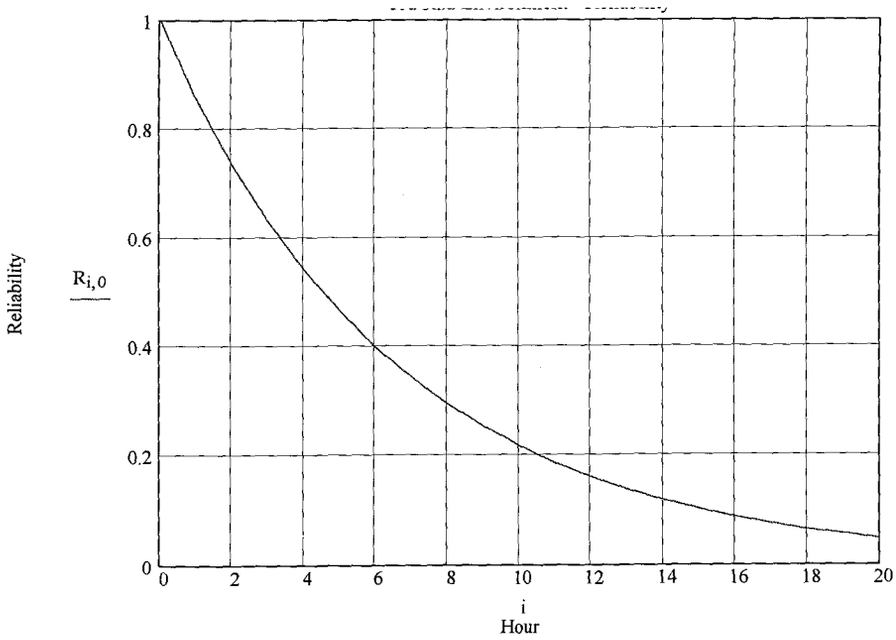

Figure 13: Estimated Reliability Over 20 Hours for Deep Space Environment, Design Case Solar Flare

Although this assumption may not be correct, it does make construction of the model tractable.

Main memory and L2 cache are protected against occurrences of single errors, and can detect two-bit errors. We only considered the effects of radiation-induced faults on the processor, including L1 cache. It is likely that this resulted in a reliability estimate higher than which would be observed under actual operating conditions. However, there was not sufficient experimental data to consider the effects of radiation-induced faults in other components (e.g. memory management unit, ALU).

4. The system software has a reliability of 1. Although this is unlikely to be the case, it simplified the construction of the model. Note that a software reliability estimate could be assigned to the transition from the "No Fault Injected" state to the "No Error" state shown in . Numerous methods of

estimating the reliability of a software system have been devised over the past several years ["Handbook of Software Reliability Engineering", ISSRE98 Paper]. It is likely that more accurate estimates of the reliability of a system which includes software implemented fault tolerance and one or more applications would result from using these techniques. However, determination of the reliability of the application or the fault tolerance software was beyond the scope of our activities.

6. Conclusions and Future Work

An initial tool set and methodology have been developed to estimate system reliability, availability and performance under mission environmental conditions. Although neither the tool set, nor the experiments are yet completed, early results are promising. Experimental results from first use indicate that COTS hardware can be used with good reliability in benign space environments such as the surface of Mars, Low Earth Orbit or Deep Space in a non-solar flare condition, without extensive fault tolerance development. With the addition of relatively straightforward fault detection and mitigation strategies, high reliability can be achieved at relatively low cost for many NASA missions. Future work will include: completion of the tool set; definition and execution of extensive fault injection campaigns encompassing both the application and the OS, and including error injection to simulate the effects of faults into uninjectable subsystems; higher fidelity system performability models; validation of the models and experimental results through space based testing.

REFERENCES

[1] R. R. Some and D. C. Ngo, "REE: A COTS-Based Fault Tolerant Parallel Processing Supercomputer for Spacecraft Onboard Scientific Data Analysis," Proc. of the Digital Avionics System Conference, vol. 2, pp. B3-1-7 - B3-1-12, 1999.

[2] J. J. Beahan, L. Edmonds, R. D. Ferraro, A. Johnston, D. Katz, R. R. Some, "Detailed Radiation Fault Modeling of the Remote Exploration and Experimentation (REE) First Generation testbed Architecture," Aerospace Conf. Proc., vol. 5, pp. 279-291, 2000.

[3] J. J. Beahan, "SWIFI: A Software-Implemented Fault Injection Tool," JPL Internal Document, June 2000.

[4] R. Some, W. Kim, G. Khanoyan, L. Callum, A. Agrawal, J. Beahan "A Software-Implemented Fault Injection Methodology for Design and Validation of System Fault Tolerance," Int. Conf. On Dependable Systems and Networks (DSN'2001), Göteborg, Sweden, July 2001

[5] R. R. Some, A, L. Callum, G. Khanoyan, J. J. Beahan, "Fault-Tolerant Systems Design – Estimating Cache Contents and Usage," IEEE Aerospace Conference, 2002

Acknowledgment

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology under a contract with the National Aeronautics and Space Administration. This project is part of NASA's High Performance Computing and Communications Program, and is funded through the NASA Office of Space Sciences.